LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-236997

Scalable Equation of State Capability

T. G. W. Epperly, F. N. Fritsch, P. D. Norquist, L. A. Sanford

December 5, 2007

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Scalable Equation of State Capability

Tom Epperly, Fred Fritsch, Peter Norquist, and Lawrence Sanford
November 30, 2007

Abstract

The purpose of this techbase project was to investigate the use of parallel array data types to reduce the memory footprint of the Livermore Equation Of State (LEOS) library. Addressing the memory scalability of LEOS is necessary to run large scientific simulations on IBM BG/L and future architectures with low memory per processing core. We considered using normal MPI, one-sided MPI, and Global Arrays to manage the distributed array and ended up choosing Global Arrays because it was the only communication library that provided the level of asynchronous access required. To reduce the runtime overhead using a parallel array data structure, a least recently used (LRU) caching algorithm was used to provide a local cache of commonly used parts of the parallel array. The approach was initially implemented in a isolated copy of LEOS and was later integrated into the main trunk of the LEOS Subversion repository. The approach was tested using a simple test, `tstcalc__Managan.c`. Testing indicated that the approach was feasible, and the simple LRU caching had a 86% hit rate.

Introduction

Because of the trend toward smaller amounts of memory per processing core, LLNL must ensure that all components of its large simulation codes scale in memory as well as processing speed. The original Livermore Equation Of State (LEOS) library, a package used widely by LLNL simulation codes, stores a complete copy of its large interpolation coefficient arrays on each processor, and this approach does not scale in memory. This techbase investigated using a parallel array data structure to improve LEOS's memory scalability.

For decades in the field of high-performance computing, we have seen growth in both the processing power and the memory available per processor. With the introduction of the IBM BG/L architecture, we see the beginning of new trend toward lower memory per processing core and no virtual memory capability on the compute nodes. Commodity CPU makers are focusing their effort now on making multi-core chips rather than continuing to focus on clock speed and better pipelining, so for the next decade we are likely to see the number of processing cores growing significantly faster than the amount of memory per board. This means that the trend observed with BG/L is likely to appear in all of the fastest supercomputers. For reasons of reliability and simplicity, compute nodes are not likely to have hard disks attached to provide virtual memory.

Simulation codes configure LEOS with a list of materials and material properties that they want to be able to calculate during the simulation. During configuration, LEOS generates a table of interpolation coefficients for each material/property pair, and then in the physics calculation modules, LEOS efficiently calculates physical properties and equation of the state information from precalculated interpolation coefficient tables.

The precalculated interpolation coefficient tables are the largest part of LEOS' memory footprint; and hence it is the data structure that we choose to address. To see the potential impact, consider a hypothetical simulation involving 20 materials, and for each material the simulation code needs 12 types of physical properties or equation of state calculations. Assume that each calculation is generated

from a table of values with 50 points in the density (ρ) dimension and 50 points in the temperature (T) dimension and that the interpolation scheme requires 12 coefficients per cell. This results in coefficient tables that consume roughly 57MB, roughly 10% of a BG/L node's total memory. With the original LEOS, each node stores a complete copy of the tables.

This techbase produced a modified LEOS that divides the coefficient tables into pieces, and each processor only stores its share of the complete coefficient table. If the total memory LEOS requires for its coefficient tables is n, the approximate memory footprint per process now scales as $\frac{n}{p}$ where p is the number of processes (normally equivalent to the number of ranks in an MPI job).

This approach essentially solves the memory scalability problem, but it introduces a runtime performance overhead because processes now have to fetch the interpolation coefficients from other processors, incurring the communication overhead of round trip communication between processes. Depending on the latency between nodes, the overhead could outweigh the benefits of using the precalculated interpolation coefficients in the first place.

However, our intuition is that most of the communication overhead can be avoided by adding a software cache on top of the parallel coefficient table data structure. Simulation calls to LEOS do not randomly sample the T-ρ space; rather, their access pattern usually follows a fairly predictable trajectory. We can use caching to retain performance while still reducing the memory footprint per process.

Requirements & Ramifications on the Approach

The first state of the project was to determine how LEOS was used and what requirements from the LEOS end users would impact the project implementation. These requirements were gathered through conversations with members of the Kull and ALE3D development teams. In this section, each key requirement is listed, and its impacts are discussed.

LEOS calls are not collective

With the exception of the initialization and finalization calls, LEOS calls are not collective. There is no attempt to coordinate calls to LEOS or to guarantee that every MPI rank calls LEOS at the same time. This requirement means that we could not use normal MPI communication or even one-sided MPI communication. Despite its name, one-sided MPI communication requires collective calls to `MPI_Win_Fence`. The Global Arrays (GA) toolkit from Pacific Northwest National Laboratory provides asynchronous, one-sided access to distributed data without explicit cooperation from the process holding the data.

LEOS may be initialized and finalized several times

During a given simulation run, LEOS may be initialized and finalized multiple times. Often a simulation starts out with an initial set of materials and types of physical properties and equation of state calculations it requires, and later, it may come to a regime where it needs to increase the list of materials. The simulations accomplish this by finalizing (freeing up LEOS resources with a call to `leos_fin_lib`) LEOS and then reinitializing it with another call to `leos_init`.

This requirement forces LEOS to use GA without its Memory Allocator (MA) subsystem. Typical GA usage requires an *a priori* upper bound on the memory to be allocated by GA. It's possible (although tedious) to calculate an upper bound on memory usage for a particular call to `leos_init`, but the

upper bound calculated during the first call to `leos_init` is not likely to be a good upper bound for the whole run given that simulations may add new materials as the simulation progresses.

Manoj Krishnan, one of the GA developers from PNNL, provided me with the undocumented (but apparently supported) procedure to build GA to use malloc and free instead of the MA subsystem. With these modifications, GA does not require the *a priori* upper bound on memory usage. For more details on building GA, see the appendix.

Multiple LEOS configurations in a simulation

Some simulations have different materials or different calculations on different processes in a large MPI run; hence, they often have LEOS configured differently on different MPI processes. In the extreme case, each individual process could have its own configuration of LEOS, but this case destroys the possibility of parallelizing the coefficient array data across multiple processor because there is no way to know *a priori* which processors can share data. Rather than coding for the extreme case, we assume that each MPI communicator (potentially a subcommunicator) has the same LEOS configuration. For the GA version of LEOS to work, all calls to `leos_init` and `leos_fin_lib` must be collective over an MPI communicator.

If different parts of the MPI run have different LEOS configurations, the simulation must call a new LEOS API routine, `leos_parallel_init`, before calling `leos_init`. This function has two arguments, the MPI communicator that defines the group of MPI processes with which this process will share data and a limit for how much memory should be used for caching array coefficients. If the client does not call `leos_parallel_init`, LEOS will assume that all the processes in `MPI_COMM_WORLD` have identifical LEOS configurations, and it will cache 1/256th of the coefficient array data locally. In my testing, 1/256th appeared to be large enough to provide good cache performance.

If the client code calls `leos_parallel_init` before the first call to `leos_init`, it must also call `leos_parallel_shutdown` after the last call to `leos_fin_lib`. `leos_parallel_shutdown` should be called before `MPI_Finalize` because it terminates GA. If the client does not explicitly call `leos_parallel_init, leos_parallel_shutdown` gets called during `leos_fin_lib.`

Minimal changes to the external LEOS API

LEOS is a long-lived program with a simple interface. It has a wide client-base inside LLNL, and LEOS end users are unlikely to adopt any changes to LEOS that require significant changes to their code.

The techbase largely succeeded in adding new capabilities without changing any of the existing API calls. It only required the addition of `leos_parallel_init` and `leos_parallel_shutdown`. We also add the constraint that calls to `leos_init` and `leos_fin_lib` must be collective over the appropriate communicator.

Minimal changes to LEOS internally

The techbase was not large enough to consider sweeping changes to the LEOS source code, so we had to adopt an approach that limited the amount of code that needed to be changed. Conceptually, the coefficient could be seen as a 5-dimensional array # materials $\times$ # functions $\times$ # density points $\times$ # temperature points $\times$ # interpolation coefficients, and the whole 5-D array could be shared as a single

parallel array distributed across all the processors in the MPI communicator. However, each material might have different functions, and the number of points in each of the remaining dimensions depends on a variety of configuration parameters.

To keep the changes to LEOS manageable and in the scope of this project, we only looked at parallelizing the 2-D and 3-D coefficient tables for each material/function pair. In the original and modified LEOS, this information is stored in a C struct called `coeff_table`. In the original LEOS, whenever they needed the interpolation coefficient information, the code directly accessed the information contained in `coeff_table`. In the modified LEOS, we added a C function API to access the coefficient table. By making the definition of `coeff_table` opaque and putting a function API between the data structure and the rest of the code, it made it possible to have two separate implementations of the interpolation coefficient array: one based on normally allocated memory and one based on GA distributed arrays.

The modified LEOS committed into Subversion

The goal of the techbase was to provide a working prototype that could be incorporated into the LEOS subversion repository. This is an important step for the LEOS developers to incorporate the changes into future LEOS releases.

Implementation

The implementation consists of two main new pieces. The first piece, the interpolation coefficient data type, simply establishes a C function API between the underlying coefficient array data types and the rest of the LEOS library. The second piece, a distributed array data type, manages the caching and provides an implementation independent API to Global Arrays.

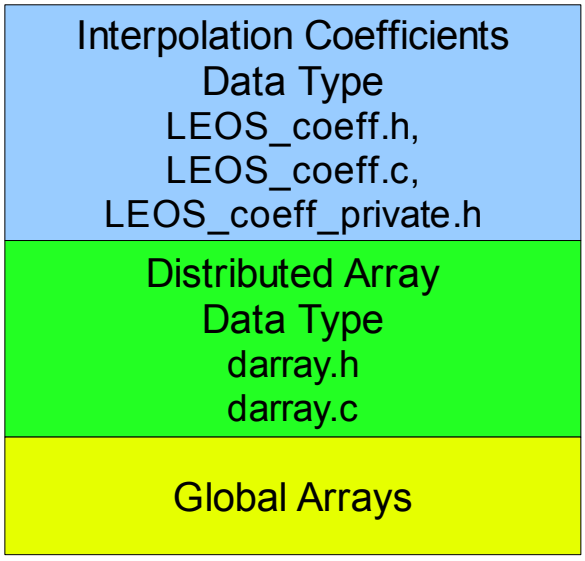

Figure 1: Implementation details

The interpolation coefficients data type has two separate implementations. First it has an implementation that works exactly like the original LEOS works. Each processes allocates enough memory to store the entire coefficient array. The second implementation uses the distributed array data type to achieve memory scalability. The user can choose which one to use when LEOS is configured.

When the project was started, it wasn't clear which technology would be used to manage the distributed array data. Hence, the distributed array data type was written to support any distributed array back-end with the required functionality. This flexibility is important if another distributed array library eclipses Global Arrays, or if a machine specific approach must be written.

One of the limitations imposed by using Global Arrays is that we lose the ability to control the fine details of how the array pieces are mapped to actual processors. On a machine like BG/L, it is important to fetch data from the nearest available source, and by using GA, we lose the ability to consider nearest neighbor optimizations. It's also worth noting that the arrays being distributed are small enough that some processors may not have any local pieces. For example, a table with 70 points

in the density dimension and 70 points in the Temperature dimension only has a total of 4900 sets of interpolation coefficients. Hence for runs involving more than 4900 processors, some processors will not be storing any array elements locally (other than cached copies). If we had direct control of the layout, we might be able to get better performance by storing several copies of the coefficient array across subsections of the communicator. Because the distributed array is write once and read many, there are not any cache coherence issues that would normally accompany this approach.

The implementation modifications do not include any changes to the code that generates the interpolation coefficient because doing so would probably require writing the whole library. Each processor calculates the whole coefficient table and temporarily stores the whole table in memory before copying it to the distributed array data structure. This process could be parallelized from a CPU and memory standpoint, but it was beyond the scope of the exploratory techbase.

In addition to the interpolation coefficients, LEOS stores the function values at each of the grid points. For a property that varies in both dimensions, the means an additional # density points $\times$ # temperature points doubles of memory. This data structure could also be distributed, but the extent of the software changes required made it infeasible to do during this techbase.

The most challenging aspect of the implementation was modifying all the places where it directly accessed the interpolation data structures to make a function call instead. As one measure of complexity, a `svn diff` from before the modifications to after the modifications has 4,741 lines[1]. The change required modifications to 26 files.

Testing

First, we tested the modified LEOS to ensure that it gave the same results as the original LEOS. To accomplish this, the output from the `tstcalc__Managan.c` test problem was compared between the original and modified LEOS. The output was consistent to the number of significant digits being printed. Second, the implementation can be compiled for debugging in which case it maintains a complete copy of the coefficient table in addition to storing the data in the distributed array. When compiled in this way, the code ensures that the values from the distributed array are bit-for-bit identical with the values stored in the local copy. These two tests both verified that the scalable LEOS was consistent with the original.

LEOS lacks a suite of comprehensive tests to validate the correctness and runtime impact of these modifications. Runtime testing was performed with `tstcalc__Managan.c` and with a trivial test problem from ALE3D provided by Jeff Keasler. Neither of these test problems accessed LEOS like a long-running multi-physics code would, so the testing was indicative rather than definitive. The time required to under stand the requirements, choose the appropriate technologies, and implement took the majority of the time allocated in the techbase. There was not enough time to perform testing with full scale multi-physics applications.

Most of the initial parallel testing took place on Thunder, and we were able to verify that the code worked correctly and that the average cache hit rate was 86%. We were also able to make some runs on UBGL (the unclassified BGL) before it was taken offline. These runs verified that the approach and the tools it depends on work correctly on BG/L architectures.

Conclusions

This techbase has demonstrated that using distributed arrays to store the LEOS coefficient table is a

1 `svn diff -r 1509:1517 libleos | wc -l` yields 4741.

feasible approach to make LEOS's memory use scalable on low memory per core architectures. It is possible to make this change with only additive changes to the LEOS API and without rewriting the library from scratch. A simple caching strategy is able to provide an 86% hit rate on simple test problems.

Comprehensive testing in large-scale multi-physics applications would be the next natural step for this work. In that context, it would be clear if the caching mechanism is sufficient to provide the required performance. If the cache is not performing adequately, it may be possible to develop a custom caching approach that includes a notion about the simulation trajectory through the phase space to provide a higher hit rate.

If higher performance message passing approach is necessary, it may be possible to implement an approach based on the ARMCI layer of Global Arrays. ARMCI provides lower level access to remote memory access.

Acknowledgments

The authors would like to thank Manoj Krishnan from Pacific Northwest National Laboratory for his help with Global Arrays. We would also like to thank the ALE3D and Kull teams for providing useful feedback in developing the requirements for this project. Lastly, we would like to thank Jeff Keasler for testing the modified LEOS with a simple ALE3D problem.

Appendix

Building Global Arrays

Global Arrays (GA) is available from Pacific Northwest National Laboratory at http://www.emsl.pnl.gov/docs/global/. To prevent GA from using its internal Memory Allocator subsystem, you must edit `ga-4-0-7/global/src/base.c` (around line 52) adjusting for the appropriate GA version. The end result should be that `AVOID_MA_STORAGE` is `#define`'d to be `1`..

```
#define AVOID_MA_STORAGE 1
```

Now configure and build GA according to the normal installation instructions in the Global Arrays User's Manual, http://www.emsl.pnl.gov/docs/global/user.html.

Building LEOS with distributed arrays

To build the memory-scalable LEOS, you must first build and install GA. When you configure libleos for building, you need to specify --with-global-arrays=<*GA prefix directory*>. This tell configure tell set the `LEOS_GLOBAL_ARRAYS` preprocessor symbol, and it adds the GA include directory to the `#include` search path. You must also specify the appropriate MPI compilers. For example, on thunder you would use:

```
./configure CC=mpiicc F77=mpiifort --with-global-arrays=<GA install prefix directory>
```

If you want to active the debugging mode where it compares the GA version of the distributed arrays versus a local `malloc`'ed copy, set the `LLD_DEBUG` preprocessor symbol when compiling. To see the cache statistics, set the `LLDDARRAY_DEBUG` preprocessor symbol. Lastly to see a cache trace, compile with `LLDDARRAY_CACHE_TRACE` set.

When linking against the modified libleos.a, simulations will also need to link against the following GA libraries:

```
-larmci -lglobal -lma
```

New LEOS API routines and Requirements

For simulations where the whole parallel computer will have the same LEOS configuration (i.e., all calls to `leos_init` and `leos_fin_lib` are collective), there are only two things that the code must guarantee to be able to use the memory scalable LEOS. First, `leos_init` must come after `MPI_Init`, and `leos_fin_lib` must come before `MPI_Finalize`. In this case, distributed array will make a cache 1/256th the size of the original coefficient array. This this approach should provide a roughly 2 orders of magnitude decrease in the memory requirements for LEOS.

For simulations where there are several LEOS configurations, the client source code will need to be modified to make a call to `leos_parallel_init` and `leos_parallel_shutdown`. The call to `leos_parallel_init` must occur after the `MPI_Init` call and before the first call to `leos_init`, and the call to `leos_parallel_shutdown` must occur after the last `leos_fin_lib` and before `MPI_Finalize`. You can also use `leos_parallel_init` to specify how much memory should be used to provide local caching for LEOS coefficient arrays. The amount specified is divided up proportionally because all the distributed arrays.

Here are the C prototypes for this new API calls:

```
/**
```

```
 * This call does not replace the call to leos_init. It should be
 * called exactly once before leos_init. If leos_init is called before
 * this subroutine, leos_init will call leos_parallel_init with
 * MPI_COMM_WORLD.
 *
 * It is assumed that this call is made collectively over the whole
 * MPI_COMM_WORLD. If you initialize LEOS identically (same set of
 * materials, functions, interpolation settings, and extrapolation
 * settings) for all processors in MPI_COMM_WORLD, the communicator
 * argument should be MPI_COMM_WORLD. However, if your simulation
 * defines subcommunicators and each subcommunicator initializes LEOS
 * differently, pass in the subcommunicator that this processor is
 * part of. This function will assume that communicator defines the
 * processor group that this processor is a part of, and this group
 * will all collectively initialize LEOS with the same arguments to
 * leos_init.
 */
Integer
leos_parallel_init(MPI_Comm communicator, /* WORLD or subcommunicator*/
                   const Integer cacheMemory);

/**
 * This routine shutdowns down the underlying technology for sharing
 * interpolation coefficients across processes. It should be called
 * exactly once before your parallel run exits. No leos property calls
 * can be made after this is shutdown.
 *
 * If leos_init called leos_parallel_init (as opposed to
 * leos_parallel_init being called by the users program before
 * leos_init was called), leos_fin_lib will call
 * leos_parallel_shutdown.
 *
 * It is doubtful whether leos_parallel_init can be successfully
 * called to reinitialize LEOS after leos_parallel_shutdown has been called.
 */
Integer
leos_parallel_shutdown(void);
```